

# Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource

Lisa R. Hsu     Steven K. Reinhardt<sup>1</sup>  
University of Michigan  
Ann Arbor, Michigan  
{hsul, stever}@eecs.umich.edu

Ravishankar Iyer     Srihari Makineni  
Intel Corp  
Hillsboro, Oregon  
{ravishankar.iyer,  
srihari.makineni}@intel.com

## ABSTRACT

As chip multiprocessors (CMPs) become increasingly mainstream, architects have likewise become more interested in how best to share a cache hierarchy among multiple simultaneous threads of execution. The complexity of this problem is exacerbated as the number of simultaneous threads grows from two or four to the tens or hundreds. However, there is no consensus in the architectural community on what “best” means in this context. Some papers in the literature seek to equalize each thread’s performance loss due to sharing, while others emphasize maximizing overall system performance. Furthermore, the specific effect of these goals varies depending on the metric used to define “performance”.

In this paper we label equal performance targets as Communist cache policies and overall performance targets as Utilitarian cache policies. We compare both of these models to the most common current model of a free-for-all cache (a Capitalist policy). We consider various performance metrics, including miss rates, bandwidth usage, and IPC, including both absolute and relative values of each metric. Using analytical models and behavioral cache simulation, we find that the optimal partitioning of a shared cache can vary greatly as different but reasonable definitions of optimality are applied. We also find that, although Communist and Utilitarian targets are generally compatible, each policy has workloads for which it provides poor overall performance or poor fairness, respectively. Finally, we find that simple policies like LRU replacement and static uniform partitioning are not sufficient to provide near-optimal performance under any reasonable definition, indicating that some thread-aware cache resource allocation mechanism is required.

## Categories and Subject Descriptors

B.3.3 [Hardware]: Memory Structures—*Performance Analysis and Design Aids*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FACT’06, September 16–20, 2006, Seattle, Washington, USA.  
Copyright 2006 ACM 1-59593-264-X/06/0009 ...\$5.00.

## General Terms

design, performance

## Keywords

cache, multiprocessor, partitioning, performance

## 1. INTRODUCTION

Several factors—including power constraints and diminishing returns on extracting instruction-level parallelism from individual threads—have led to a major industry shift toward chip multiprocessors (CMPs). While some early CMPs employed private per-core cache hierarchies, more recent designs including Intel’s Core Duo [6], IBM’s Power 5 [9] and Sun’s Niagara [11] use shared last-level on-chip caches.

Shared caches enable more flexible and dynamic allocation; in the extreme, one core can use the entire cache space if the other cores are idle. Shared caches can also provide higher performance when cores share data. A single copy of a cache block can be accessed from all cores with low latency, reducing coherence traffic. The reduction in data redundancy also leads to larger effective capacity on chip. This effect can potentially offer significant performance improvements over private caches, particularly in larger server platforms where the workloads are known to share data between threads [12, 11]. Additionally, server threads from the same application often run the same binary, thus sharing instructions, and virtual machines running the same guest operating system can read from the same operating system binaries [20]. All of these forms of sharing have the potential to benefit from shared caches.

As with all resources being shared by competing elements without regulation, there is the danger of destructive interference, unfairness, or starvation of individual threads. There is already evidence of such problems in current hyper-threaded systems where two threads share L1 and L2 caches, in which a garbage collection thread in a database application running at the same time as worker threads created such destructive interference that the performance of the database suffered dramatically [4].

This phenomenon demonstrates the need for a policy to control the cache allocation process, which in turn raises the question of what the optimal policy might be. Of course, the

<sup>1</sup>Also with Reservoir Labs, Inc.

answer to this question hinges on the objective for which we optimize. Do we seek to maximize overall system performance, or fairness across threads? Given one of these goals, what metric do we seek to maximize or equalize? While the notion of performance in single-threaded uniprocessors is straightforward, multithreaded systems are amenable to multiple definitions, including IPCs and miss rates, both raw (absolute) and weighted—and among weighted metrics, multiple weighting factors may be selected. The goal of this paper is to explore the impact of various definitions of “optimal performance” on the partitioning of a shared cache among multiple threads. Specifically, we seek to answer the following questions:

- To what extent does the definition of “optimal” impact the resulting partitions and performance?
- If we target one definition of “optimal”, how well does the system fare with regard to other reasonable definitions? In particular, how do policies targeting overall performance rate in terms of fairness, and how do policies targeting fairness rate in terms of overall performance?
- If we define optimality based on metrics that are not readily available online, are there online measurable metrics that correlate well that could be used to drive an online policy?

Borrowing loosely from economics terminology, we define three high-level policies for cache allocation. A “Communist” approach seeks to maximize fairness, ensuring that each thread bears an equal portion of the cache sharing penalty, or alternatively derives an equal benefit from the presence of the cache. The goal of a “Utilitarian” policy is to maximize the total benefit for the aggregate group—e.g., by maximizing total throughput—without regard to individual thread performance. Finally, a “Capitalist” cache policy is an unregulated free-for-all—the most common policy in use today.

Intuitively, either a Communist or Utilitarian policy seems preferable to a Capitalist policy, as both of these seek to maximize some desirable property, while the Capitalist policy makes no effort toward any form of “goodness”. However, the issue of which metric to apply to a Utilitarian or Communist policy remains. Miss rate is a common metric for cache performance, but miss rate is not always proportional to perceived performance. Memory bandwidth is a scarce commodity in CMPs, so it may be appropriate for a Utilitarian policy to minimize total bandwidth. Throughput is often the bottom line, but raw IPC is not a very useful metric, since different threads have different levels of ILP. Weighted IPCs [14] are attractive and have been used in past studies, but raise the question of what weighting factor should be used.

Overall, we identify four distinct aspects of a cache partitioning policy: *performance targets*, *evaluation metrics*, *policies*, and *policy metrics*.

- A *performance target* is the end goal. A performance target may be for all threads to have the same miss rate, or that all threads should have the same IPC relative to some baseline configuration. The performance targets we examine are minimizing total memory bandwidth, maximizing weighted IPC with respect

to various baselines, and equalizing weighted IPC with respect to various baselines.

- The *evaluation metric* is the metric used to express the performance target and to evaluate the extent to which the target is achieved. For example, if the target is to maximize total weighted IPC, then the evaluation metric is weighted IPC.
- A *policy* is the aspect of the cache implementation which makes allocation decisions. One well-known policy is LRU replacement, intended to achieve low miss rates in the general case. We compare two simple policies—LRU and uniform static partitioning—to policies that perform allocation based on metric values measured from application behavior. Because we are investigating fundamental behavior rather than specific implementations, we use static offline optimal partitioning to model these policies.
- A *policy metric* is a metric used to drive an allocation policy (the offline optimal policies in our case). Ideally, the policy metric and evaluation metric would be identical. However, the desired evaluation metric may not be measurable online, and thus may not be useful in driving policy decisions. Practical implementations may be forced to use policy metrics that are online observable metrics that merely correlate with the evaluation metric.

We find that the optimal partitioning of a shared cache can vary greatly as different but reasonable definitions of optimality are applied. We also find that, although Communist and Utilitarian targets are generally compatible, each policy has workloads for which it provides poor overall performance or poor fairness, respectively. Weighted metrics, which may be difficult to measure online, correlate strongly with their unweighted counterparts for Utilitarian targets, and thus the latter may be useful as proxies for performing online decision-making; however, Communist targets have no such appropriate proxies. Finally, we find that simple policies like LRU replacement and static uniform partitioning are not sufficient to provide near-optimal performance under any reasonable definition, reinforcing that a thread-aware cache resource allocation mechanism is required.

We begin the remainder of the paper by discussing related work in Section 2, our experimental methodology (Section 3), followed by our results (Section 4). We conclude in Section 5 and offer our plans for future work in Section 6.

## 2. RELATED WORK

Since CMPs are a relatively new development on the processor scene, the amount of related work is relatively small. Most previous work focuses on using miss rates as performance targets in SMT systems.

Stone et al. [15] developed a model for studying the optimal allocation of cache memory among multiple access streams. They experimentally determine a miss rate curve that maps cache size to miss rate for a reference stream, and then fit that curve to an exponential function. Noting that the total miss rate for a pair of memory access streams is the average of the two contributing miss rates, they point out that solving for a minimal miss rate merely involves taking the derivative of this equation, setting it to zero, and solving. They also showed that LRU typically comes close to

achieving optimal performance. They focused on partitioning a cache between the instruction and data access streams of a single workload, and did not consider partitioning across multiple workloads.

Thiebaut et al. [19] build on Stone’s work to partition disk caches for maximal hit ratios. They utilize shadow tags, which are tags without data, to indicate hits that could have occurred had there been a larger allocation. Using this information, they calculate the marginal gain of adjusting the cache allocation. They note that a problem with implementing a greedy marginal gain algorithm with this methodology is actually finding the memory stream with the largest marginal gain, since the functions are non-monotonic. They resort to performing a sort every time they update a partition. Their study assumed fully associative disk caches and partitioning on a disk block granularity. This amount of computation is likely too hefty for a CMP, while it is acceptable for a long latency entity like a disk cache.

Suh et al. [18] propose a partitioning scheme that depends on offline profiling of applications. With the knowledge of offline miss rates, an online partitioning unit determines an optimal partition for minimizing miss rates. They later developed a fully online scheme involving additional counters in the hardware [16, 17] to calculate the marginal miss rate reduction for each additional cache way for a given thread, which they find is a reasonable approximation for each additional block if the cache has sufficient associativity. This mechanism is tied to a set-associative LRU cache and limited to partitioning the cache on a per-way granularity rather than per block. They use a greedy algorithm to allocate cache partitions once they determine the marginal gains.

Chiou et al. [2] propose a partitioning scheme that partitions at the granularity of cache ways. Partitioning is achieved by limiting the cache ways in which a thread can place its data. The flexibility in placement is thus limited.

Kim et al. [10] present a cache partitioning algorithm which focuses on fairness in a small scale CMP using SPEC2000 benchmarks. They evaluate several metrics and correlate them to execution time to determine what a good online metric to drive their policy decisions should be, and develop an algorithm that attempts to keep those metrics as equal as possible throughout execution time.

Chandra et al. [1] deal not with partitioning but with predicting inter-thread cache contention in a shared cache on a CMP, with the intent to use this information to prevent thrashing in a shared cache. They present a mechanism to accurately predict when threads will thrash, though they do not present a means to prevent it.

Huh et al. [5] focus on providing flexibility in setting cache sharing levels in the hardware. The platform they present is a NUCA platform, and the dynamic extent of their research is on mapping blocks to an appropriately close cache bank. Their emphasis is on hardware implementation.

Iyer [8] presents a framework for providing differentiated services to various threads via the cache hierarchy in a CMP. The framework consists of classifying heterogeneous memory streams, assigning priorities, and enforcing them. He presents several means of enforcing a partition, including selective allocation and set partitioning.

Fedorova et al. [3] present an operating system scheduling algorithm to deduce which sets of threads would coexist the best to schedule at the same time. Their goal is to schedule threads which would yield the lowest overall miss rates

without starving any threads. This technique may not be relevant on large scale CMPs where the number of software threads may not outnumber the number of hardware threads by so much as to make scheduling an issue. Furthermore, large-scale CMPs will likely support multiple virtual machines, making system-wide optimization outside the scope of any one OS scheduler.

### 3. METHODOLOGY

In this section we describe our performance target selection, the method used to obtain optimal allocations for these performance targets, the benchmarks used, and the CMP thread model.

#### 3.1 Performance Target Selection

A performance target is some function of the overall cache allocation which we seek to minimize or maximize by adjusting the allocation. We use  $(p_1, p_2, p_3, \dots, p_N)$  to denote the set of cache allocations for an  $N$ -thread workload, such that  $p_i$  is the cache allocation for thread  $i$  and  $\sum_{i=1}^N p_i = C$ , where  $C$  is the total cache space available.

As discussed previously, targets can be either Communitist or Utilitarian in nature, emphasizing either fairness or overall performance. Each of these high-level targets can be applied to a variety of metrics.

A metric has two components: a “base” metric and an (optional) weighting factor. We consider three base metrics:

- *Misses per access (MPA)*, i.e., miss rate. This is a fundamental cache performance metric that is easy to measure online.
- *Misses per cycle (MPC)* measures bandwidth usage. We include this metric because off-chip bandwidth may be a precious resource in future servers.
- *Instructions per cycle (IPC)* is instruction rate. For a fixed program path length, e.g. no spin-wait loops, IPC directly corresponds to throughput, and is thus a direct measure of observed performance.

A meaningful performance target should reflect some tangibly useful goal. Because IPC directly indicates performance, we focus primarily on IPC-based targets. We also consider MPC as it may prove useful for severely bandwidth-constrained systems. We do not consider MPA-based performance targets, since low miss rates do not directly translate into measurable improvement on the system level. Nevertheless, MPA is attractive because of its ease of online measurement; we will return to MPA in Section 4.4 to consider whether it is useful as a policy metric, i.e., a measurable proxy for other more significant metrics.

The other component, the weighting factor, was born out of SMT processor performance studies [14]. Any of the metrics defined above can be weighted, i.e.:

$$Metric_{weighted}(p_i) = \frac{Metric(p_i)}{Metric(baseline)} \quad (1)$$

The need for weighting came about because of the observation that simply maximizing raw aggregate IPC leads to policies that favor inherently high-IPC threads at the expense of low-IPC threads. Using weighted IPCs, where the baseline is the IPC a thread achieves when it has the processor to itself, eliminates this bias, yielding a metric that does

Target Name	Description
MPC-None-Ut	Minimizing Overall Bandwidth
IPC-128-Ut	Maximizing IPC weighted wrt 128KB performance
IPC-256-Ut	Maximizing IPC weighted wrt 256KB performance
IPC-512-Ut	Maximizing IPC weighted wrt 512KB performance
IPC-1024-Ut	Maximizing IPC weighted wrt 1MB performance
IPC-128-Co	Equalizing IPC weighted wrt 128KB performance
IPC-256-Co	Equalizing IPC weighted wrt 256KB performance
IPC-512-Co	Equalizing IPC weighted wrt 512KB performance
IPC-1024-Co	Equalizing IPC weighted wrt 1MB performance

**Table 1: Performance targets.**

not reward starvation of low-IPC threads. This same concept can be used when comparing policies for cache resource sharing.

For cache sharing studies, the choice of baseline for weighted metrics is less clear. In the context of a CMP system with 64 cores and 32MB of shared cache, the performance of a single thread when it has the system to itself (i.e. with all 32MB of cache) is largely irrelevant to its performance with the roughly 512KB of cache it would be expected to receive in normal operation. This baseline would also be costly to measure online, as it would require idling 63 cores for long enough for a single thread to warm up such a large cache, then repeating this task 64 times for each running thread. Instead, we choose a set of baseline partition sizes that bracket the static uniform allocations for our systems—128KB, 256KB, 512KB, and 1024KB—to see the impact of baseline selection on the resulting partitioning.

A useful performance target must combine a metric with a Utilitarian or Communist model to describe some meaningful optimization goal. A Utilitarian target minimizing raw MPC seeks to alleviate the demand for potentially limited off-chip bandwidth, and thus may be useful. However, we do not see any motivation for MPC targets based on weighted MPC or a Communist model. As discussed above, raw IPCs also do not make meaningful performance targets due to their tendency to favor high-IPC threads at the expense of low-IPC threads, so we use only weighted IPC (WIPC) for our performance targets. We apply both the Utilitarian model (maximizing WIPC, i.e., minimizing aggregate relative degradation) and the Communist model (equalizing WIPC, i.e., equalizing relative degradation across all threads). Table 1 lists all the performance targets evaluated in this study.

### 3.2 Optimal Cache Allocations

Having selected a set of performance targets, the question turns to determining an optimal allocation for a given target. Because we are studying the fundamental behavior of different performance targets, we use static offline analysis to determine the optimal partition for a given target. We start by discussing partitioning using MPA-based targets, then use an analytic model to extend this methodology to the MPC- and IPC-based targets we desire.

Past studies have observed that the overall cache miss rate (measured in terms of misses per access, or MPA) is a simple function of the sum of the miss rates of the contributing threads [15, 18]. That is, for  $N$  threads, where  $MPA_i(p_i)$

is the the miss rate of of thread  $i$  with cache allocation  $p_i$ :

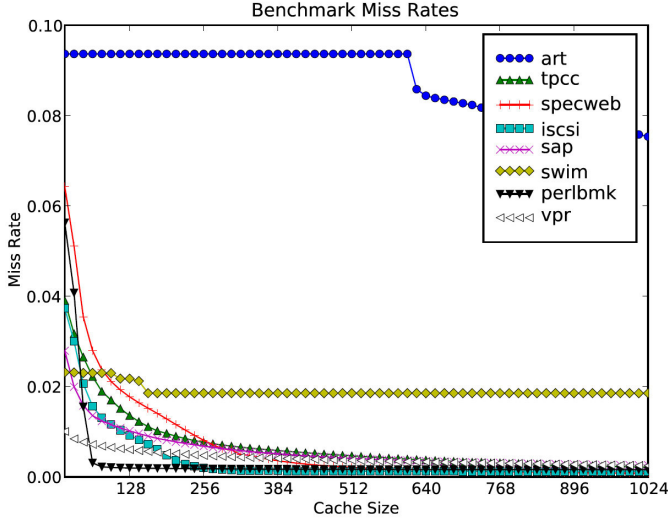
$$OverallMissRate = \frac{1}{N} \sum_{i=1}^N MPA_i(p_i) \quad (2)$$

Combining this observation with miss rate data for each application of interest on a range of cache sizes, a simple search algorithm can statically determine the optimal cache allocation  $p_1, p_2, \dots, p_N$  for applications 1 –  $N$  that minimizes the total misses incurred.

To obtain the needed miss rate data, we utilize a trace-based behavioral cache simulator called CASPER [7] to determine the miss rate functions of several benchmark applications for a spectrum of cache sizes. The CASPER simulator yields non-timing cache behavior information like miss rate, and is parameterizable according to size, associativity, block size, and split versus unified. We use CASPER to obtain miss rate curves for eight benchmarks from cache sizes 16KB to 1024KB in increments of 16KB; results are shown in Figure 1. We limited our cache size to 1MB because our data showed that miss rate errors would have been unacceptably high with larger cache sizes. Error rates stem from the fact that in trace-based simulation, the first miss to a block may have been a hit had the simulation been running from the beginning, rather than from the middle of a trace [21]. So, all cold misses are actually of unknown status and comprise the error margin. Error margins can be reduced by running longer simulations, but we were limited by the lengths of some of the traces we were able to obtain, and instead limited our simulated cache sizes in order to ensure reasonable error margins.

We simulated caches with 64 byte blocks and as close to 32-way associativity as possible. Since some cache sizes were not perfect powers of 32 (e.g., 48KB), 32-way associativity would be impossible. In cases like these, we used the nearest associativity to 32; in the case of 48KB, it is 24-way associativity. We used high associativity because we did not want conflict misses to unduly color our results for small cache sizes.

We selected eight benchmarks for our studies. Since CMPs are likely to be useful in the server market, we chose some common server workloads: TPC-C, SpecWeb, iSCSI, and SAP. At the same time, we wanted a variety of miss rate behaviors, and thus selected some other benchmarks from the SPEC2000 suite. Art and swim are what we consider to be “pathological” workloads, in the sense that no amount of cache will reduce miss rates. We wanted workloads of this type to mimic the pathological behavior observed in current



**Figure 1: Miss rate curves for the benchmarks used in this study. On the x-axis is cache size in KB, and on the y-axis is miss rate in terms of misses per access MPA.**

systems [4]. On the other end of the spectrum are perlbnk and vpr, meant to represent threads that may not be particularly cache intensive. From the graph, it is clear that we got what we had hoped for in terms of a range of miss rates and a range of “knee” behavior. Traces ranged from 40 million to 200 million instructions in length. The TPC-C and SAP traces are Intel traces and are the shorter traces, while the SpecWeb and iSCSI server traces were taken from a full-system M5 simulator [13] simulation. The SPEC2000 server workload traces were also taken from the M5 simulator, but in a non-full-system mode simulation.

Since we also want to study MPC and IPC as metrics, we note that we can use the same static offline methodology as long as we have a representation of cache size-to-IPC/MPC mappings. To get these mappings, we use an analytical model to determine MPC and IPC from the MPA data we measured using CASPER. For MPC (bandwidth), we require a function which translates  $MPA(p)$  to  $MPC(p)$ . In the following equations,  $MPI$  is misses per instruction,  $API$  is accesses per instruction,  $CPI_{base}$  is the base CPI of the CPU, and  $CPI_{memory}$  is the CPI added due to miss penalties.

$$MPC = \frac{MPI}{CPI}$$

$$MPC(p) = \frac{MPA(p) * API}{CPI_{base} + CPI_{memory}}$$

$$MPC(p) = \frac{MPA(p) * API}{CPI_{base} + MPA(p) * API * MissPenalty} \quad (3)$$

Thus, Equation 3 determines MPC as a function of cache size with a given MPA curve.  $API$  is constant for each benchmark, and experimentally determined. Since our traces were taken from both RISC and CISC sources, we adjusted  $API$  for the Intel traces to reflect accesses per micro-op, so

that numbers between the traces would be comparable. We set  $CPI_{base}$  to 1, and  $MissPenalty$  to 500. We use a large miss penalty because we are modeling last level cache miss penalties (i.e. going to DRAM); in addition, we wanted to emphasize the impact of the memory system in our experiments, since we are focused on the qualitative rather than quantitative behavior of cache sharing policies.

Our model for IPC is similar, using the same constants:

$$IPC = \frac{1}{CPI}$$

$$IPC(p) = \frac{1}{CPI_{base} + CPI_{memory}}$$

$$IPC(p) = \frac{1}{CPI_{base} + MPA(p) * API * MissPenalty} \quad (4)$$

Note that we use the somewhat more awkward IPC form rather than CPI because CPI is not additive across threads.

Converting these equations to weighted metrics is trivial; a thread’s performance at a given cache size is merely divided by the thread’s performance at the baseline size.

Given all of these size-to-metric curves, determining allocations that optimize over the aggregate or equalize across contributing threads is a simple matter. For Utilitarian targets, we consider all possible partitions and choose the partition that yields the best overall performance. For Communist targets, we again consider all possible partitions and choose the partition that yields the lowest standard deviation between all contributing threads. The equations below use weighted IPC with respect to 256KB as an example.

To find the optimal partition for IPC-256KB-Utilitarian, we merely have to maximize:

$$WIPC_{total}(p_1, p_2, \dots, p_N) = \sum_{i=1}^N \frac{IPC_i(p_i)}{IPC_i(256KB)} \quad (5)$$

For IPC-256KB-Communist, we minimize over all threads  $i$ :

$$\sigma = \text{stddev}\left(\frac{IPC_i(p_i)}{IPC_i(256KB)}\right) \quad (6)$$

### 3.3 CMP Thread Model

In our experiments, we model 2, 4, 8, 16, and 32 threads sharing 1MB of cache. We construct the workloads by choosing every possible two- and four-thread application mix from our eight benchmarks, with our larger workloads created by logical replication, e.g., a 2-application mix is replicated 4 times to yield an 8-thread workload, while a 4-application mix is replicated twice.

Due to our static trace-based methodology, a four-thread workload consisting of two threads from application A and two threads from application B sharing an  $N$  MB cache yields the same partition for each thread as a two-thread workload consisting of one A thread and one B thread sharing an  $N/2$  MB cache. Thus we can determine the partition for a 32-thread workload on a 1MB cache by evaluating the constituent four-thread workload on a 128KB cache. This technique shortens our search time significantly since we now only need to evaluate all possible ways to divide 128KB between four threads, rather than all possible ways to divide 1MB by 32 threads.

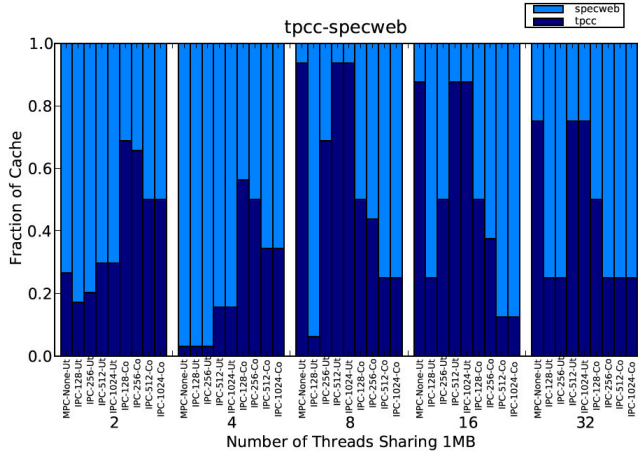


Figure 2: TPC-C and SpecWeb sharing a CMP platform can yield widely varying ideal allocations, depending on the definition of ideal.

## 4. RESULTS

We begin our analysis by presenting the impact of the chosen performance target on the allocation of cache capacity among threads. We then focus on two specific aspects of performance target selection: Communist vs. Utilitarian models and the choice of baseline when using weighted metrics. This section ends by looking at two issues related to implementation feasibility: whether there are policy metrics measurable online that correlate well with desired evaluation metrics, and whether simple policies such as LRU and static partitioning can approximate any reasonable performance target.

### 4.1 Optimal Partitions

We statically determined the optimal partitions on all possible two- and four-application mixes for 2 to 32 threads for the various performance targets we examine. We found that cache allocations can vary widely depending on the target.

Figure 2 shows an example of how significantly partitioning can vary due to different definitions of optimality. This particular graph shows the various optimal cache allocations of the TPC-C/SpecWeb application mix. The clusters along the horizontal axis represent different numbers of threads sharing a 1MB cache. Each member of a cluster is a different performance target. The first component of the target indicates the evaluation metric, MPC or IPC. For IPC targets, the numerical component of a performance target indicates the weighting factor, e.g. IPC-128 is IPC weighted against IPC(128KB). The -Ut or -Co suffix indicates whether it is a Utilitarian or Communist target. The vertical axis indicates the fraction of cache given to each benchmark. For example, in the case of 16 threads and the IPC-512-Ut target, TPC-C receives about 90% of the 1MB cache, meaning that the 8 threads of TPC-C receive about 90% of the cache, or equivalently that a single thread of TPC-C is about 90% of 128KB. As is clear from the graph, the allocations can vary widely even from within a metric “family”, like IPC-\* -Ut or IPC-\* -Co. This graph shows evidence that Communist targets can differ significantly from their Utilitarian counterparts, and that even choosing different weighting factors

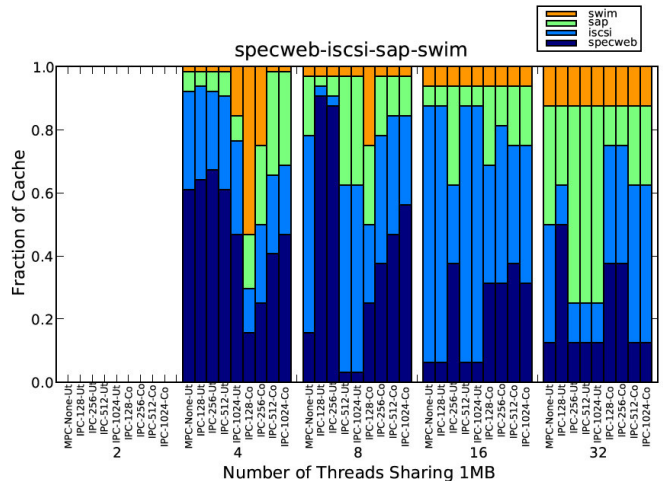


Figure 3: Another example of widely varying partitions, with four benchmarks sharing a CMP. The 2 thread column is empty because there is no way to run 4 benchmarks on only 2 threads.

can yield significantly different results.

Figure 3 shows the same results for the 4-application case. Optimal partition behavior varies widely and the impact of changing performance targets is difficult to predict. Trends caused by varying the baseline cache size are non-monotonic. For example, at 16 threads, going from IPC-128-Ut to IPC-256-Ut yields a larger partition for SpecWeb, but its partition size shrinks again when moving from IPC-256-Ut to IPC-512-Ut.

To quantify how much partitions can vary overall, we devised a partition difference metric. For any application mix, we determine the average difference in allocation for a single thread when going from any one performance target to another. Specifically, say the optimal partition for  $App_A$ -

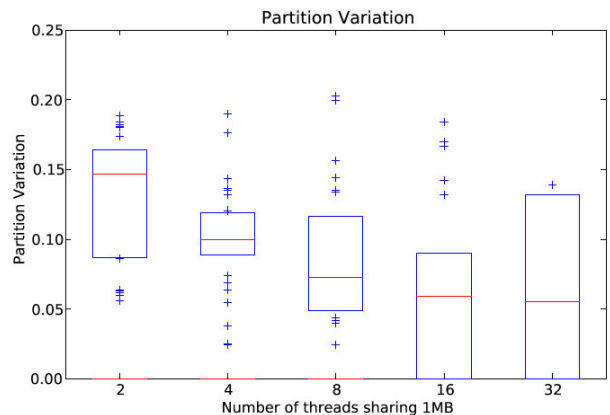
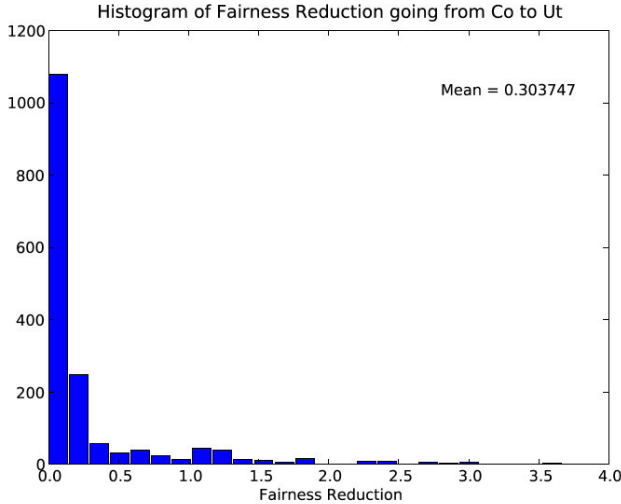


Figure 4: The variation in partition size between any given pair of performance targets. The values on the y-axis indicate the average relative change in cache allocation as the performance target changes.



**Figure 5: Histogram of the absolute reduction in fairness (i.e. absolute differences in  $\sigma$  when going from a Communist target to its Utilitarian counterpart (e.g. IPC-Co-128 to IPC-Ut-128)).**

$App_B$  with performance target  $i$  is such that  $App_A$  receives allocation  $p_{iA}$  and  $App_B$  receives allocation  $p_{iB}$ . Then the partition difference metric is defined as:

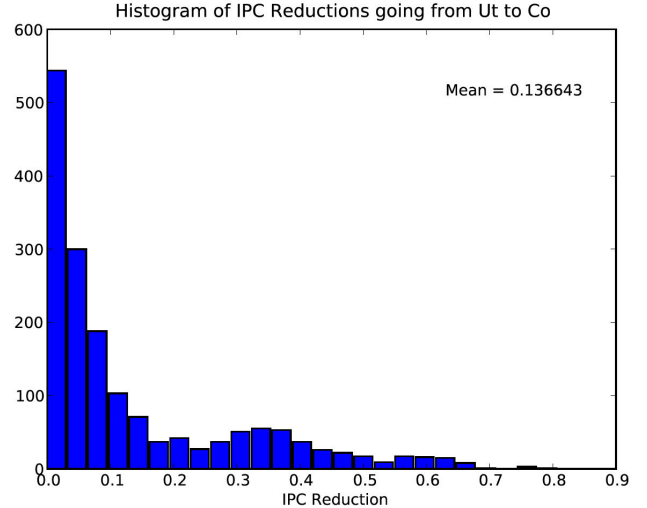
$$\frac{\sum_i \sum_{j>i} \sum_x |p_{ix} - p_{jx}|}{x_{max} * \binom{i_{max}}{2}} \quad (7)$$

The numerator represents the sum of all the allocation changes between all possible pairs of performance targets and all pairs of applications within the mix, while the denominator represents the total number of possibilities, with the result yielding an average. Figure 4 shows the spectrum of partition variation for all the mixes used in this study. This box-and-whiskers plot shows the inner two quartiles inside the box, with the horizontal line representing the median. The whiskers extending outside the box show the remaining values of the outer two quartiles. The y-axis shows the change in allocation relative to a static uniform partitioning.

Partition variation between performance targets when two threads share 1MB can vary by almost 20%, with the median at 15%. The 16- and 32-thread cases many zero-change instances resulting in lower medians because the 16KB partitioning granularity we use represents a significant fraction of the typical per-thread allocation in a 1MB cache. However, there are still cases where there is a significant partition variation between different performance targets. In all cases, it is clear that varying the performance target selection among a set of reasonable choices can yield significant variations in the optimal cache partition.

## 4.2 Communism versus Utilitarianism

One of the key distinctions among previously proposed performance targets is whether they target overall performance (Utilitarian) or fairness (Communist). Both of these goals are desirable, and it is difficult to say in general which is more important. In this section, we evaluate the quantita-



**Figure 6: Histogram of the percentage reduction in weighted IPC when going from a Utilitarian target to its Communist counterpart.**

tive difference between these targets. Specifically, we look at how much fairness is lost when using a Utilitarian target in place of a Communist target (Figure 5), and conversely, how much throughput is lost when using a Communist target in place of a Utilitarian target (Figure 6).

In each case, we look at all pairs of workloads sharing the same weighted IPC metric, and generate histograms indicating how far the objective function for one model deviates from its optimal value when the partition is optimized for the other model. (We ignore the raw MPC metric used previously since it does not make sense under a Communist policy.) For example, the IPC-Ut-128 target provides maximal throughput using weighted IPC with a 128KB baseline as the evaluation metric (maximizing  $WIPC_{total}$  of Eq. 5, while IPC-Ut-128 uses the same metric but optimizes for fairness (minimizing the  $\sigma$  of Eq. 6). The delta between the  $\sigma$  values of these models is plotted in Figure 5, while the delta between the  $WIPC_{total}$  values is plotted in Figure 6.

Both histograms have most of their samples in the bin closest to zero, meaning that for the most part, Communist and Utilitarian metrics are comparable; i.e. optimizing for throughput tends to provide near-optimal fairness and vice versa. However, both distributions have a significant tail. In other words, Communist targets mostly yield partitions that have near-optimal overall performance, but there are a few cases where the optimal Communist partition yields extremely poor overall performance. Likewise, Utilitarian targets mostly yield optimal partitions that are not extremely unfair, but there are a few cases which are.

These two graphs do not indicate which target is preferable, but they do indicate that whichever target is used, it is important to guard against these heavy tails such that the resultant partitions can be both fair and have good overall performance. The better starting point remains to be seen and would depend on ease and feasibility of implementation.

## 4.3 Baseline Weighting Choices

As previously discussed, one of the subtleties to choosing

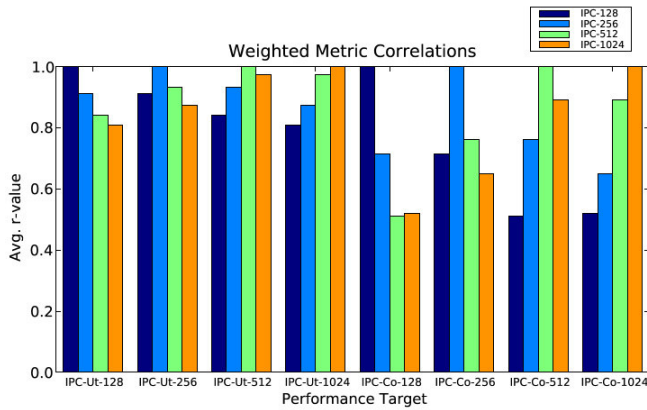


Figure 7: This graph demonstrates the dramatic differences there can be in choosing different baselines for weighted metrics. Each cluster has a bar with value 1 because every metric correlates perfectly to itself.

a weighted performance target is the selection of the baseline weighting factor. Prior work has taken the baseline as “what the performance would be if it had the whole system to itself”. While this approach may be reasonable for small-scale systems, it may be both impractical and irrelevant on an LCMP system with tens or hundreds of threads and tens of megabytes of cache.

Figure 7 examines the impact of the choice of baseline by showing the correlation coefficients between weighted metrics with various baselines. A perfect correlation of 1.0 indicates that there is no difference between choosing one baseline or another. Lower correlation values indicate larger differences between the results of different baselines. As can be seen from the graph, Utilitarian weighted IPC metrics are all reasonably close, with correlations staying above 0.8. However, with Communist weighted IPCs, the selection of weighting factor has a great impact on resulting performance. Given this sensitivity to the weighting factor, it seems difficult to define “fairness” in a robust and precise fashion.

#### 4.4 Policy Metrics

As mentioned in the introduction, an evaluation metric may not be measurable online in real-time. A weighted metric is difficult to measure online because knowing how some thread would fare if it had some greater portion of the cache is impossible to know without offline profiling, which can be inaccurate when inputs vary, and typically does not capture time-varying program behavior, or online sampling, which may be impractical if it requires running each of many threads alone sequentially in order to achieve unperturbed samples. Thus, it is useful to find what we term *policy metrics*, online measurable metrics that serve as proxies for evaluation metrics and can be used to drive policy decisions that yield good performance with respect to the performance target. In this section, we study the correlations between metrics that are reasonably measurable online and our selected performance targets. We consider unweighted metrics to be measurable, as they can typically

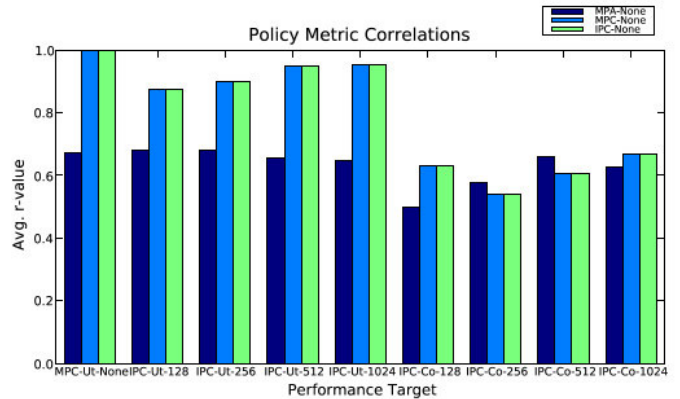


Figure 8: Correlations between possible policy metrics and performance targets. The y-axis shows the absolute value of r-values.

be captured online using simple counters without requiring special sampling phases.

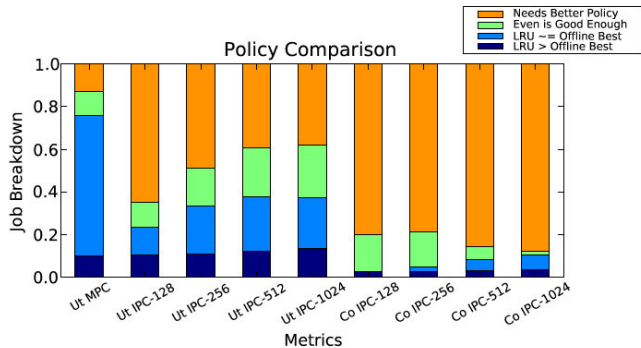
The results are shown in Figure 8. The graph shows that miss rate (MPA) is actually correlates quite poorly with all performance targets. Making partition decisions using MPA as a replacement for the performance targets of interest would not yield desired results. MPC and IPC, however, are relatively good indicators for Utilitarian targets, but poor for Communist targets. This result indicates that there is no sufficient policy metric that we know of to drive an online policy for Communist weighted IPC targets. For the benchmarks we have chosen, choosing to target a Communist policy is futile because there is no way to measure how well you are doing. Thus it may make more sense to use Utilitarian targets, while simply trying to avoid the unfair outliers seen in the previous section.

One interesting side result is that MPC (bandwidth) and IPC always correlate perfectly. Upon further investigation, we discovered that this is an artifact of our performance model described in Section 3.2. In this model, raw MPC and raw IPC are linearly related, such that minimizing raw MPC yields the same partitions as maximizing raw IPC. We expect that this relationship will continue to hold, though less precisely, in real-world executions.

#### 4.5 Policy Evaluations

Given the difficulties in defining “optimal” behavior for cache partitioning, we revisit the initial hypothesis that some explicit cache management policy is required to provide good behavior in a shared-cache system. Is it possible that simple, existing policies such as LRU provide adequate cache management under some objective function?

We answer this question by comparing two simple policies, LRU and uniform cache partitioning, with static offline optimal partitioning. The goal is to show the general sensitivities of these optimal policies, and whether simpler mechanisms like LRU or or even partitioning are sufficient. Figure 9 shows that, unlike in uniprocessor scenarios, LRU is not particularly close to optimal for any of our performance targets. On the x-axis are the various performance targets, while on the y-axis the combination of all application mixes are broken down into four categories. Each appli-



**Figure 9:** This graph compares the performance LRU and static uniform partitioning (even split) policies with our candidate performance targets.

cation mix has an evaluation metric value according to the performance target it was being optimized for. That same evaluation metric is measured under the other two policies, LRU and evenly distributed partitioning. The bottom two bars represent cases where LRU’s evaluation metric performance measures within 10% of optimal, i.e. LRU is a sufficient policy for reaching near-optimal performance. As can be seen, LRU does a poor job for every target except raw bandwidth (aka raw IPC). This indicates something we already knew—LRU does a generally good job for maximizing raw IPC [15]. The next bar up shows cases where LRU does a bad job, but uniform cache distribution remains within 10% of optimal. And finally, the top bar shows cases where an online policy is required to get near optimal.

The graph leads us to conclude that for Utilitarian policies, there are two options. One is to always have a policy actively pursue optimal partitions, since previous experiments show that is possible to do online using IPC or MPC as a policy metric. However, an online policy like this could prove costly, leading to the other possibility of using LRU most of the time, and have some monitoring scheme determine when even partitioning or an online policy is required.

For Communist policies, however, it is clear that even in the general case some active policy is required. Unfortunately, the previous set of experiments showed that an online policy searching for optimal will be more difficult to implement than Utilitarian policies.

## 5. CONCLUSIONS

Controlling the allocation of shared cache resources among threads is a necessary task in future CMP systems. However, the selection of the objective function for guiding this allocation is a subtle issue, and different seemingly reasonable objectives can lead to significantly different results.

We make the distinction between *performance targets*, the goals of a cache; *evaluation metrics*, the metrics to measure the achievement of that goal; *policies*, the decision rules meant to achieve the performance target; and *policy metrics*, the actual metrics used to direct policy decisions. Many researchers use policy metrics and evaluation metrics interchangeably in the literature. However, our results indicate that a common policy metric, miss rates, is a poor proxy for

the real targets of interest. Additionally, selecting different performance targets can cause optimal cache allocations to vary greatly, which implies that the selection of a performance target cannot be done in an arbitrary manner.

We compare the effect of applying Communist or Utilitarian goals in cache partitioning, i.e., optimizing for fairness across threads or for optimal aggregate behavior. We find that, for most workloads, there is little conflict between these targets, in that Communist goals typically lead to near-global optima, and Utilitarian goals typically provide good fairness. However, in either case, there are outlying workloads for which a Communist goal severely degrades global performance or a Utilitarian goal severely impacts fairness. Thus choosing one or the other of these goals as primary may be less useful than guaranteeing that neither fairness nor global performance is unduly sacrificed in pathological conditions.

We also show that weighted metrics, a standard technique in the area of SMT research intended to alleviate unfairness in resource allocation, have unexpected issues as well. CPU-oriented SMT research uses standalone performance—i.e., a single thread having all resources to itself—as a baseline. However, using standalone performance as a baseline for a multi-megabyte shared cache in a large-scale CMP is both less practical and less meaningful. Unfortunately, choosing different arbitrary baselines yields significant variations in performance, particularly for Communist targets, and it is unclear why there might be any reason to pick one over another.

Additionally, we investigate the ability of online measurable policy metrics to act as proxies for more desirable but less practical evaluation metrics. We show that, for the benchmarks selected, miss rate is a poor policy metric, since it does not correlate well with any desirable the performance targets. However, raw IPC and MPC are relatively good policy metrics for weighted IPC evaluation metrics in the Utilitarian model. We did not find a good policy metric for Communist performance targets.

Finally, we determined that optimal performance can be relatively sensitive, such that simple established methods like LRU or uniform static partitioning are insufficient for achieving near-optimal performance. LRU does a good job of maximizing raw IPC. However, LRU does not typically come within 10% of optimal for Utilitarian weighted IPC targets, and performs even more poorly for Communist WIPC targets. Even uniform static partitioning does little good when LRU does poorly. Thus, particularly in Communist models, we find that an online policy to manage cache allocation for optimal performance is necessary to achieve good performance the shared caches of CMPs.

## 6. FUTURE WORK

The conclusions of this work lead to a number of natural next steps. One thing we were interested in is injecting prioritization into our performance target definitions as a way of providing differentiable quality of service (QoS). This study has mostly dealt with the special case where all threads are equally important in the eyes of the optimization.

Another aspect ripe for future work is to analyze what characteristics caused LRU to perform poorly. Since an online algorithm searching for optimal may be very costly, it may be better to have an algorithm which uses LRU while monitoring for the characteristics that cause LRU to per-

form poorly, and switching to a different policy when those characteristics are detected. Along a similar vein, we would like to know how different “natural” partitions are from optimal ones. In other words, in a Capitalist system, how does the cache self-allocate, and how does this compare with an allocation determined by an optimal-seeking policy? This can give us clues to the general sensitivity of optimal, as well as possible identifiers that indicate when LRU will perform poorly.

Additionally, obvious future work entails developing a dynamic online mechanism for determining and enforcing optimal partitions. Since we have discovered that Utilitarian and Communist targets are mostly comparable, we will investigate implementing both targets (including heavy-tail guarding mechanisms), with an eye on which is more easily implemented or more feasible. This work will leave the trace-based simulation arena and move to cycle-accurate, and eventually, full-system simulations. We hope to create an end-to-end cooperative system between the hardware, hypervisor, and operating system for cache management.

## Acknowledgments

This work was supported by a grant from Intel and a Lucent Fellowship.

## 7. REFERENCES

- [1] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proc. 11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 340–351, Feb. 2005.
- [2] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *Proceedings of Design Automation Conference, Los Angeles*, June 2000.
- [3] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proc. 2005 USENIX Technical Conference*, pages 395–398, 2005.
- [4] R. Goodwins. Does hyperthreading hurt server performance? [http://news.com.com/Does+hyperthreading+hurt+server+performance/2100-1006\\_3-5965435.html?tag=nefd.top](http://news.com.com/Does+hyperthreading+hurt+server+performance/2100-1006_3-5965435.html?tag=nefd.top), Nov. 2005.
- [5] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A nuca substrate for flexible cmp cache sharing. In *Proc. 2005 Int'l Conf. on Supercomputing*, pages 31–40, 2005.
- [6] Intel Corp. Next leap in microprocessor architecture: Intel core duo. White paper. [http://ces2006.akamai.com.edgesuite.net/yonahassets/CoreDuo\\_WhitePaper.pdf](http://ces2006.akamai.com.edgesuite.net/yonahassets/CoreDuo_WhitePaper.pdf).
- [7] R. R. Iyer. On modeling and analyzing cache hierarchies using CASPER. In *Proc. 11th Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 182–187, Oct. 2003.
- [8] R. R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *Proc. 2004 Int'l Conf. on Supercomputing*, pages 257–266, 2004.
- [9] R. Kalla, B. Sinharoy, and J. M. Tendler. Ibm power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, Mar. 2004.
- [10] S. Kim, D. Chandra, and Y. Solihin. Fair caching in a chip multiprocessor architecture. In *Proc. 13th Ann. Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 111–122, Sept. 2004.
- [11] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, March/April 2005.
- [12] S. R. Kunkel, R. J. Eickemeyer, M. H. Lipasti, T. J. Mullins, B. O’Krafka, H. Rosenberg, S. P. VanderWiel, P. L. Vitale, and L. D. Whitley. A performance methodology for commercial servers. *IBM Journal of Research and Development*, 44(6):851–871, November 2000.
- [13] M5 Development Team. The M5 Simulator. <http://m5.eecs.umich.edu>.
- [14] A. Snaveley and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proc. Ninth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 234–244, Nov. 2000.
- [15] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Computers*, 41(9):1054–1068, Sept. 1992.
- [16] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *Proc. 2001 Int'l Conf. on Supercomputing*, pages 1–12, 2001.
- [17] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proc. 8th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2002.
- [18] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic cache partitioning for simultaneous multithreading systems. In *Proc. 13th IASTED Int'l Conference on Parallel and Distributed Computing Systems*, 2001.
- [19] D. Thiebaut, H. S. Stone, and J. L. Wolf. Improving disk cache hit-ratios through cache partitioning. 41(6):665–676, 1992.
- [20] C. A. Waldspurger. Memory resource management in vmware esx server. In *Proc. 2002 USENIX Technical Conference*, pages 181–194, Dec. 2002.
- [21] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proc. 1991 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 79–89, May 1991.